The slides uses this semester are derived from slides originally prepared by the textbook authors, Randall Bryant and David O’Hallaron.

They were modified with permission and reformatted for use in our class.

Abstraction vs. Reality

Abstraction is good, but don’t forget reality!

Most courses to date have emphasized abstraction.

- Abstract data types!
- Asymptotic analysis!

These abstractions have limits!

- Especially in the presence of bugs!
- Need to understand underlying implementations!
- Need to have a working understanding of architecture!

- Theme of the course
- Five great realities of computer science
- How this class fits into the CS curriculum
**Desired Outcomes**

Useful outcomes!
- Become more effective programmers!
  - Able to find and eliminate bugs efficiently!
  - Able to tune program performance!
- Prepare for later systems classes: Compilers, Operating Systems, Networks, Computer Architecture, Embedded Systems, many others.

**Hint:** Hang onto your book. You’ll be using this same book (3rd edition) in CS439.

---

**Great Reality 1**

**Ints are not Integers; Floats are not Reals.**

Is $x^2 \geq 0$? For floats, yes. For ints, not necessarily.

$$40000 \times 40000 \rightarrow 1600000000$$

$$50000 \times 50000 \rightarrow ??$$

Is $(x + y) + z = x + (y + z)$?

For int’s: yes. For floats, maybe not.

$$(1e20 + -1e20) + 3.14 \rightarrow 3.14$$

$$1e20 + (-1e20 + 3.14) \rightarrow ??$$

---

**Experiment**

Get into the habit of writing programs to experiment with the architecture:

```c
void main() {
    printf("40000 * 40000 = %d\n", 40000 * 40000);
    printf("50000 * 50000 = %d\n", 50000 * 50000);
    printf("1e20 + (-1e20 + 3.14) = %f\n", 1e20 + (-1e20 + 3.14));
    printf("(1e20 + -1e20) + 3.14 = %f\n", (1e20 + -1e20) + 3.14);
}
```

> gcc tester.c
> a.out

40000 * 40000 = 1600000000
50000 * 50000 = -1794967296
1e20 + (-1e20 + 3.14) = 0.000000
(1e20 + -1e20) + 3.14 = 3.140000

---

**Computer Arithmetic**

Computer arithmetic does not generate random values. Arithmetic operations have important mathematical properties.

But you cannot assume the “usual” properties of arithmetic.
- Due to finiteness of representations.
- Integer operations satisfy ring properties: commutativity, associativity, distributivity.
- Floating point operations satisfy ordering properties: monotonicity, values of signs.

**Observation:**
- Need to understand which abstractions apply in which contexts.
- Important issues for compiler writers and serious application programmers.
Great Reality 2

You've got to know assembly!

You won't often program in assembly. Compilers are much better at it and more patient than you are.

Understanding assembly is key to machine-level execution models.

- Behavior of programs in presence of bugs; high-level language model breaks down.
- Tuning program performance and understanding sources of program inefficiency.
- Implementing system software
  - Compiler has machine code as target
  - Operating systems must manage process state
- Creating / fighting malware: x86 is the language of choice for attackers.

Diving Down to Assembler Level

There are hardware resources that are not accessible from C or other high level languages.

```c
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;

/* Set *hi and *lo to the high and low order bits of the cycle counter. */

void access_counter(unsigned *hi, unsigned *lo) {
    asm("rdtsc; movl %%edx, %0; movl %%eax, %1"
    : "=r" (*hi), "=r" (*lo)
    : "%edx", "%eax");
}
```

This is a C program, with embedded x86 assembler.

Great Reality 3

Memory Matters!

Memory is not unbounded!

- It must be allocated and managed.
- Many applications are memory dominated.

Memory referencing bugs especially pernicious. The effects may be distant in both time and space.

Memory performance is not uniform.

- Cache and virtual memory effects can greatly affect program performance.
- Adapting your programs to characteristics of memory system can lead to major speed improvements.

Memory Referencing Bug Example

```c
double fun(int i) {
    int a[2];
    double d[1] = {3.14};
    a[i] = 1073741824; /* Out of bounds reference */
    return d[0];
}
```

Assume x86 (double is 8 bytes; int is 4 bytes). This will be different on other systems, and may cause segmentation fault on some.

<table>
<thead>
<tr>
<th>Call</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun(0)</td>
<td>3.14</td>
</tr>
<tr>
<td>fun(1)</td>
<td>3.14</td>
</tr>
<tr>
<td>fun(2)</td>
<td>3.1399998664856</td>
</tr>
<tr>
<td>fun(3)</td>
<td>2.00000061035156</td>
</tr>
<tr>
<td>fun(4)</td>
<td>3.14, then segmentation fault</td>
</tr>
</tbody>
</table>
Memory Referencing Bug Explanation, Little Endian

```c
double fun(int i)
{
    int a[2];
    double d[1] = {3.14};
    a[i] = 1073741824; /* Out of bounds reference */
    return d[0];
}
```

**Modified Call Result**

- `a[0] fun(0) → 3.14`
- `a[1] fun(1) → 3.14`
- `d_3...d_0 fun(2) → 3.1399998664856`
- `d_7...d_4 fun(3) → 2.00000061035156`
- `saved state fun(4) → 3.14, then seg fault`

What can you infer about how the memory is laid out?

**Memory Referencing Errors**

- **C and C++ do not provide any memory protection.**
  - Out of bounds array references
  - Invalid pointer values
  - Abuses of malloc/free

**This can lead to nasty bugs.**

- Whether or not bug has any effect depends on system and compiler.
- **Action at a distance**
  - Corrupted object logically unrelated to one being accessed.
  - Effect of bug may be first observed long after it is generated.

**How can I deal with this?**

- Program in Java, Lisp, or ML
- Understand what possible interactions may occur
- Use or develop tools to detect referencing errors

---

Memory Performance Example

The following is a matrix copy example:

```c
/* ij */
for (i=0; i<n; i++)
{
    for (j=0; j<n; j++)
    {
        b[i][j] = a[i][j];
    }
}
```

**Fact:** Memory accesses are expensive, so your operating system fetches a contiguous “block” of memory and stores that in the cache, to make subsequent accesses faster.

This one computes precisely the same result.

```c
/* ji */
for (j=0; j<n; j++)
{
    for (i=0; i<n; i++)
    {
        b[i][j] = a[i][j];
    }
}
```

But the performance is very different (several times slower on a Pentium 4), particularly for large arrays. Can you guess why that may be?
There’s more to performance than asymptotic complexity.

Constant factors matter too!
- Even an exact op count does not predict performance.
- Easily see 10:1 performance range depending on how code is written.
- Must optimize at multiple levels: algorithm, data representations, procedures, and loops.

Must understand the system to optimize performance.
- How programs are compiled and executed.
- How to measure program performance and identify bottlenecks.
- How to improve performance without destroying code modularity and generality.

Computers do more than execute programs.

They need to get data in and out. The I/O system is critical to program reliability and performance.

They communicate with each other over networks. Many system-level issues arise in the presence of networking.
- Concurrent operations by autonomous processes
- Coping with unreliable media
- Cross platform compatibility
- Complex performance issues

Course Perspective

Most systems courses are “builder-centric.”
- Computer Architecture: Design pipelined processor in Verilog.
- Operating Systems: Implement large portions of operating system.
- Compilers: Write compiler for simple language.
- Networking: Implement and simulate network protocols.

This course is programmer-centric.
- The purpose is to show how by knowing more about the design of the underlying system, one can be more effective as a programmer.
- Enable you to
  - Write programs that are more reliable and efficient
  - Incorporate features that require hooks into OS: concurrency, signal handlers, etc.
- Not just a course for dedicated hackers. We bring out the hidden hacker in everyone.
- Cover material in this course that you won’t see elsewhere.